CS255: Computer Security

Dynamic Testing

Chengyu Song 01/26/2022

Finding Vulnerabilities

- From attackers perspective
  - Vulnerabilities are the pass into the system
  - 0-day vulnerabilities are especially valuable
    - $5k ~ $1.5M, even higher in blackmarket
    - Bug bounties
- From defender/vendors perspective
  - Finding and fixing bugs before release is way cheaper than patches

Finding Vulnerabilities

How?

- Two general approach: static analysis and dynamic analysis
- Static analysis does not execute the program
  - Example: compiler warnings and errors
  - Sacrifice accuracy for code coverage -> no false negative
- Dynamic analysis performs the analysis while running the program
  - Example: unit tests
  - Sacrifice code coverage for accuracy -> no false positive

Dynamic Testing

- Unit/regression tests
  - Goal: target code behave as **expected**
  - How: (mostly) manually generated test cases
- Exploits
  - Inputs that can trigger **unexpected** behaviors
- Fuzzing and Symbolic Execution
  - Finding such inputs

Dynamic Testing

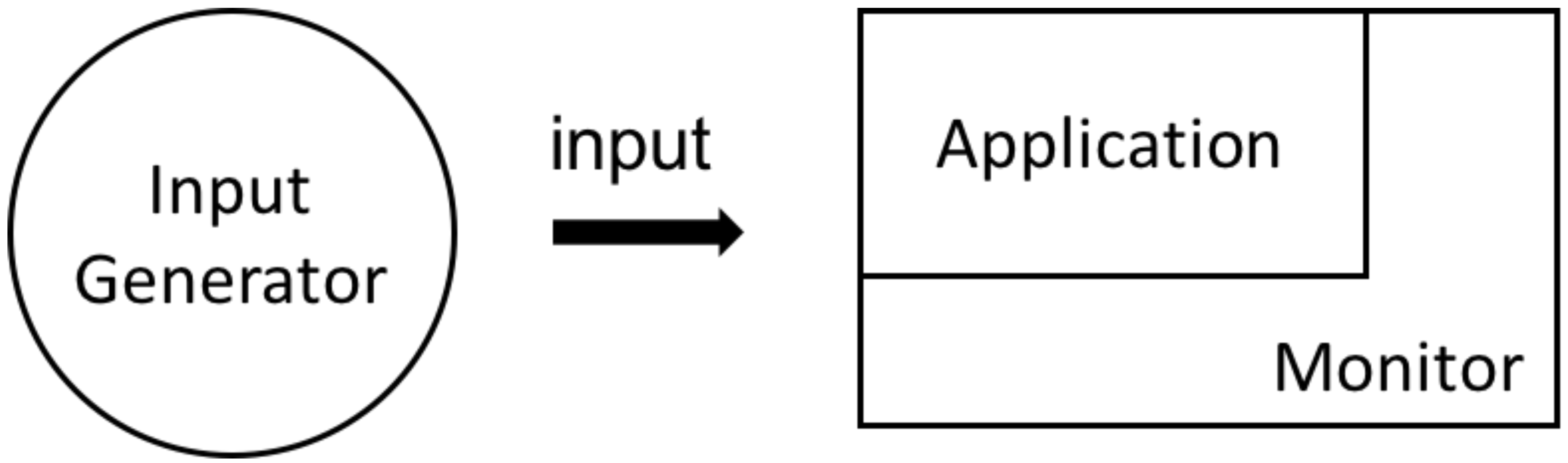
Regression vs. Fuzzing

	Regression	Fuzzing
Definition	Run program on many **normal** and known bad inputs, look for badness.	Run program on many **abnormal** inputs, look for badness.
Goals	Prevent **normal user** from encountering errors.	Prevent **attackers** from discovering exploitable errors.

Fuzzing

Three main components

- Input generator: **automatically** generates test inputs
- Executor: executes the target program with inputs
- Monitor: detects **abnormal** behaviors



Fuzzing

How to generate inputs?

- Idea 1: randomly

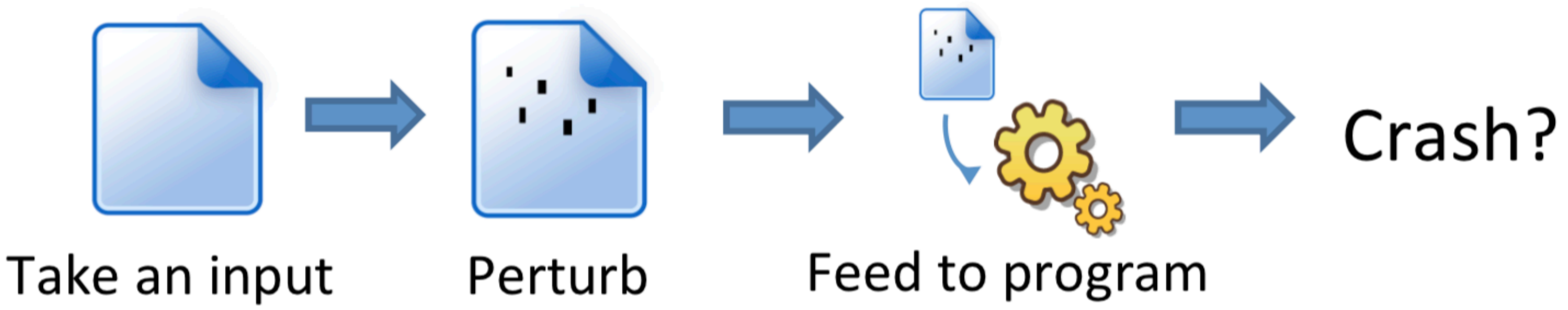
Fuzzing

Random inputs

- Advantages: easy to implement, do not look like normal inputs
- Disadvantages: inefficient
  - Programs usually have input validation logic, random inputs are unlikely to pass
  - No indication of progress

Fuzzing
Random mutation

- Idea 2: take a well-formed input, randomly perturb (e.g., flipping bits)
    - Little or no knowledge of the structure of the inputs is assumed
    - Anomalies are added to existing valid inputs
    - Anomalies may be completely random or follow some heuristics (e.g. remove NUL, shift character forward)



Fuzzing

Building a PDF fuzzer

- Google for PDF files (`filetype:pdf` more than 1 billion results)
- Crawl them to build a corpus
- Using a fuzzing tool (or script)
    1. Grab a file
    2. Mutate the file
    3. Feed it to the PDF reader
    4. Look for crash

Fuzzing

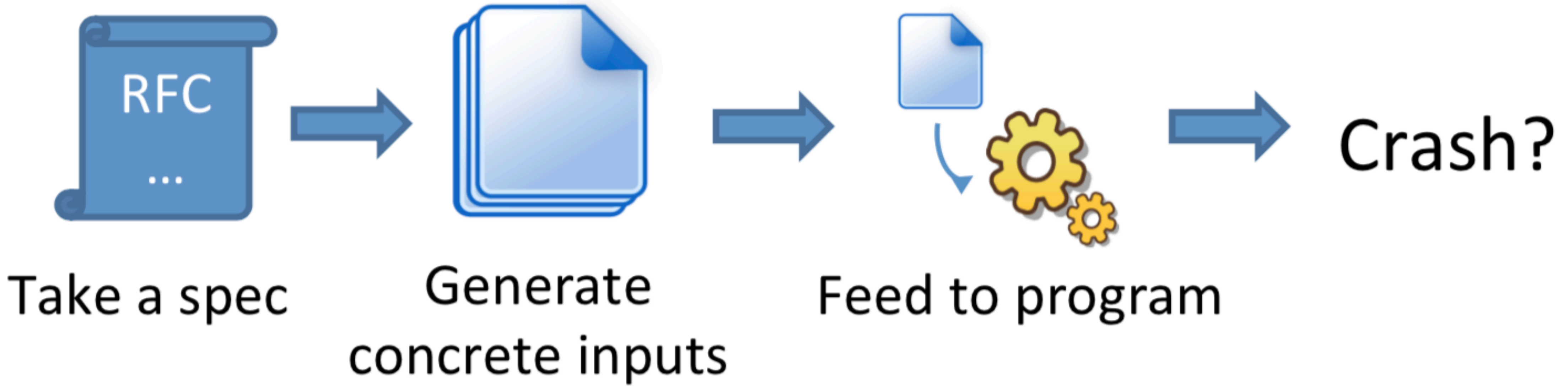
Limitations of our simple PDF fuzzer

- Only as good as the initial corpus
- Corpus may contain lots of redundant or dull inputs
  - Solution: corpus distillation
- Not making use of semantic information

Fuzzing

Syntax-guided input generation

- Test cases are generated from description of the format: grammar, RFC, documentation, etc.
- Anomalies are added to each possible spot in the inputs
- Knowledge of syntax should give better results than random fuzzing



Fuzzing

Example: png specification

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, based on colortype
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Fuzzing

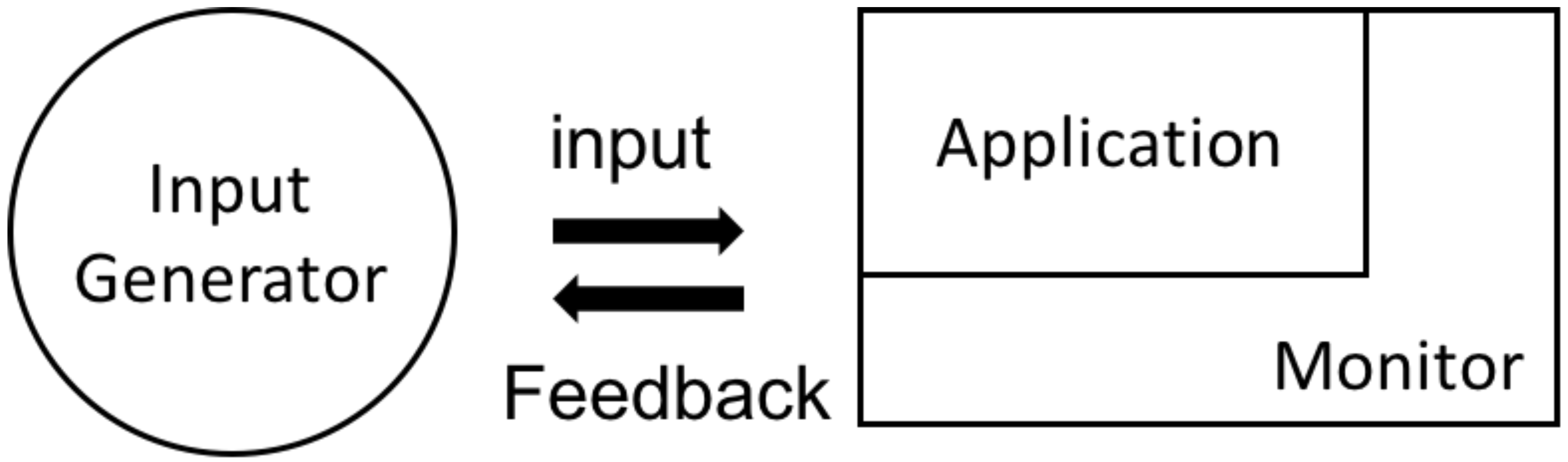
Limitations of syntax-guided generation

- Writing specifications and corresponding generators are not easy, especially for complex format
    - Solution: learn spec through machine learning
- May be too well-formed

Fuzzing

Measuring progress

- Q: with limited computation resources, how to evaluate inputs and prioritize good ones?
- Feedback fuzzing



Fuzzing

Genetic programming

Inspired by biological evolution and its fundamental mechanisms, Genetic Programming software systems implement an algorithm that uses *random mutation, crossover, a fitness function*, and multiple generations of evolution to resolve a user-defined task.

— GeneticProgramming.com

Fuzzing

Fitness function

- How to we measure the “fitness” of an input?
- Metrics
  - Program states coverage
  - Code coverage: line, branch, path, etc.
- Why? Rarely exercised code is more likely to have unknown bugs

Fuzzing

Line coverage

- Line/block coverage: measures how many lines of source code have been executed
- For the code below, how many test cases (pairs of (a,b)) is needed to achieve full (100%) line coverage?

```
if (a > 2) a = 2;
if (b > 2) b = 2;
```

Fuzzing

Branch coverage

- Branch coverage: measures how many branches in code have been taken (conditional jumps)
- For the code below, how many test cases (pairs of (a,b)) is needed to achieve full branch coverage?

```
if (a > 2) a = 2;
if (b > 2) b = 2;
```

Fuzzing

Path coverage

- Path coverage: measures how many execution paths have been taken
- For the code below, how many test cases (pairs of (a,b)) is needed to achieve full path coverage?

```
if (a > 2) a = 2;
if (b > 2) b = 2;
```

- How to calculate the total number of paths?

Fuzzing

Problems of code coverage metrics

```
mySafeCpy(char *dst, char* src) {
  if(dst && src)
    strcpy(dst, src);
}
```

- Can full line coverage guarantee to find the bug?
- Can full branch coverage guarantee to find the bug?
- Can full path coverage guarantee to find the bug?
  - What’s wrong with path coverage?

Fuzzing

Mutation strategies

- Random mutation
- Feedback-based mutation: sub-task
  - Checksum
  - Magic number
  - Likelihood to trigger vulnerability

Fuzzing

Monitors

- How to we know if there is bug?
  - Crash?
  - Manually inserted assertions?
  - **Error detectors**
    - AddressSanitizer, ThreadSanitizer, MemorySanitizer, UndefinedBehaviorSanitizer, DataFlowSanitizer, LeakSanitizer

Fuzzing

Best available fuzzer?

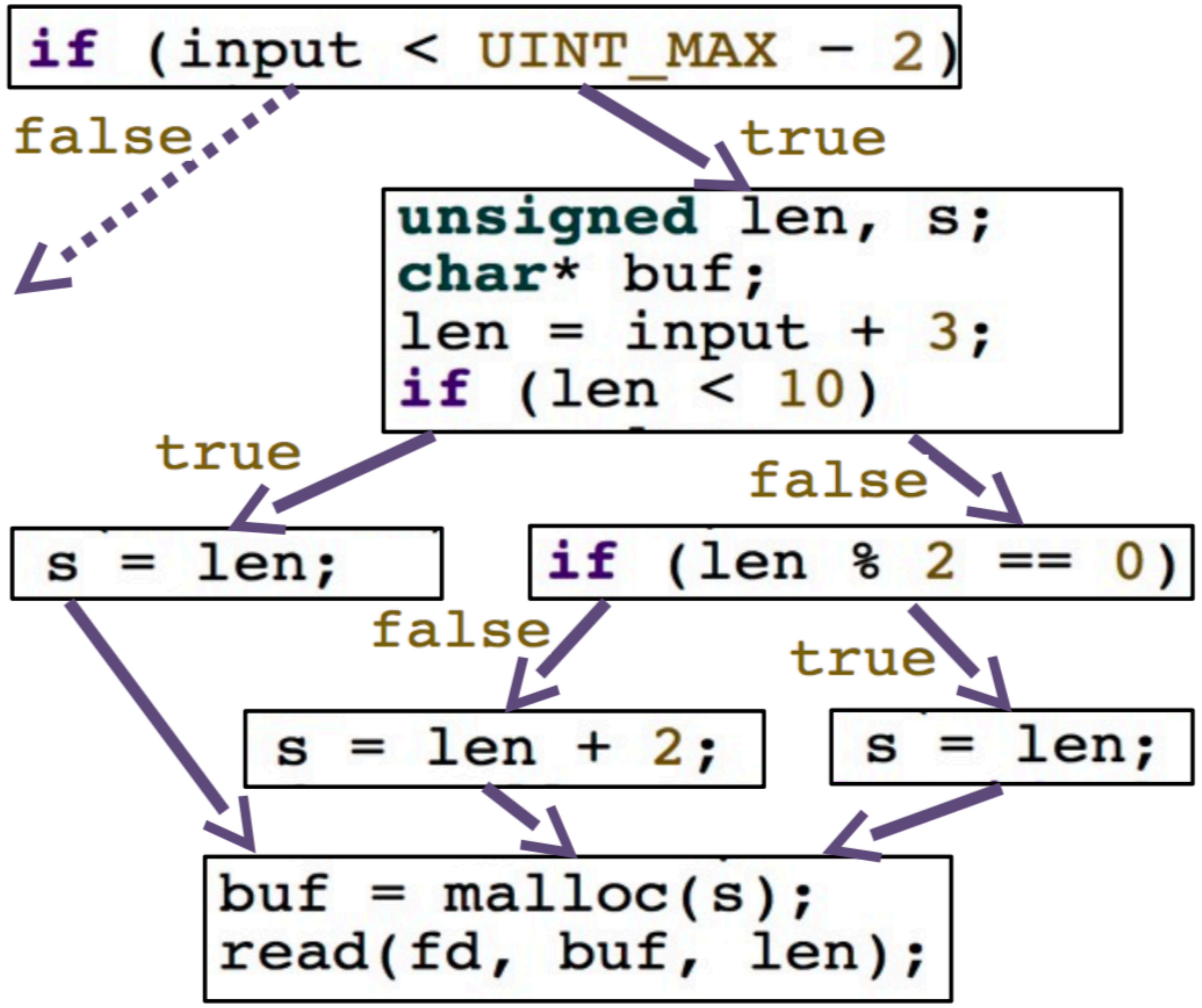
- AFL++: https://aflplus.plus/
- libFuzzer: https://llvm.org/docs/LibFuzzer.html
- honggfuzz: https://github.com/google/honggfuzz
- libafl: https://github.com/AFLplusplus/LibAFL

Symbolic Execution

Quiz: coverage

- What is the number of lines, branches, and paths?

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
      ....
  }
}
```

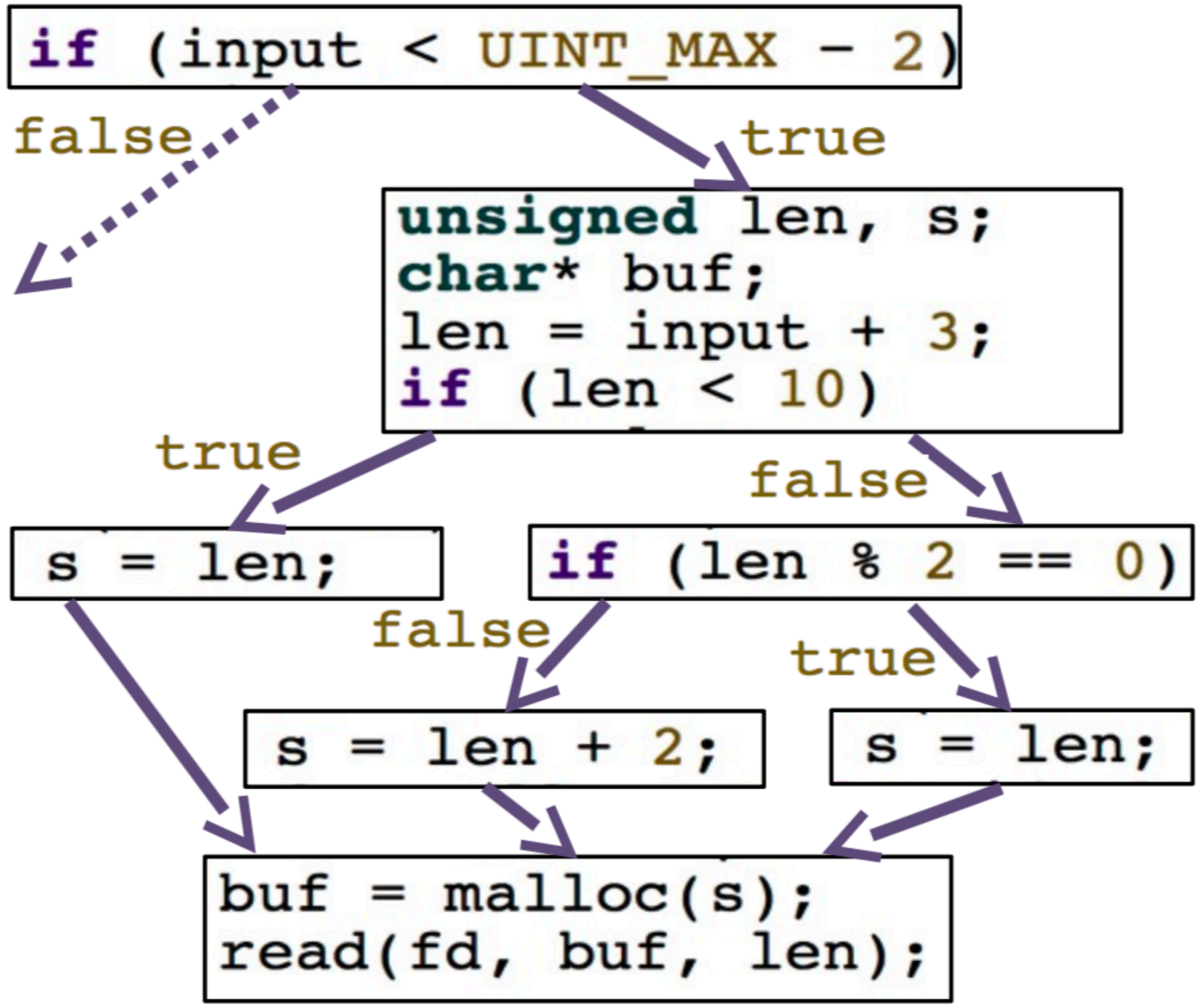


Symbolic Execution

Quiz: inputs for full coverage

- How many inputs are required for full lines, branches, and paths coverage?

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```

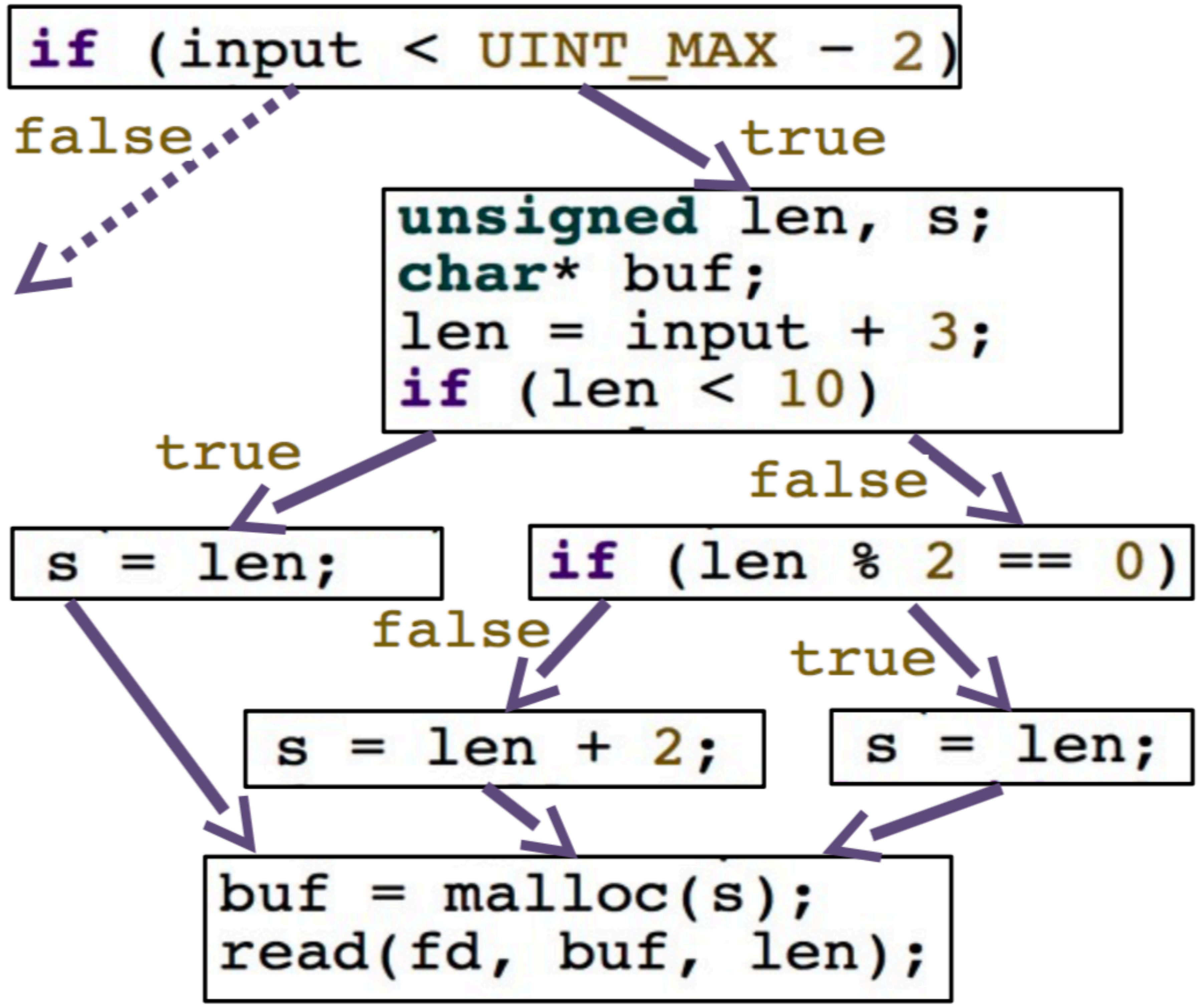


Symbolic Execution

Quiz: bug triggering input

- What is the expected number of inputs required to cover the highlighted line, using random test-case generation? Assuming unsigned is 32

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```



Symbolic Execution

Efficiency of testing

- We can evaluate the efficiency of an input generation technique using the following formula

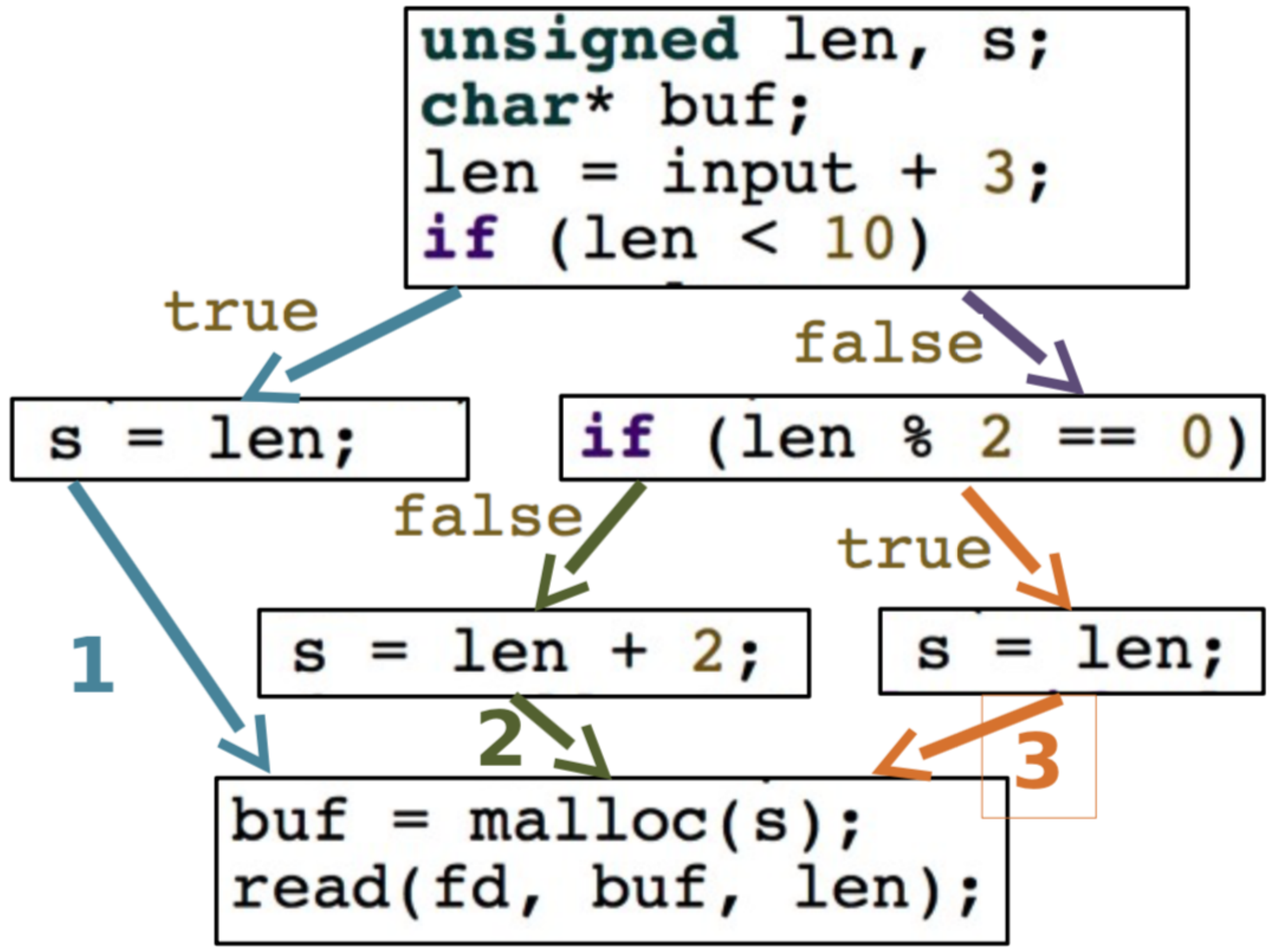
```
minimum # of inputs / expected # of inputs
```

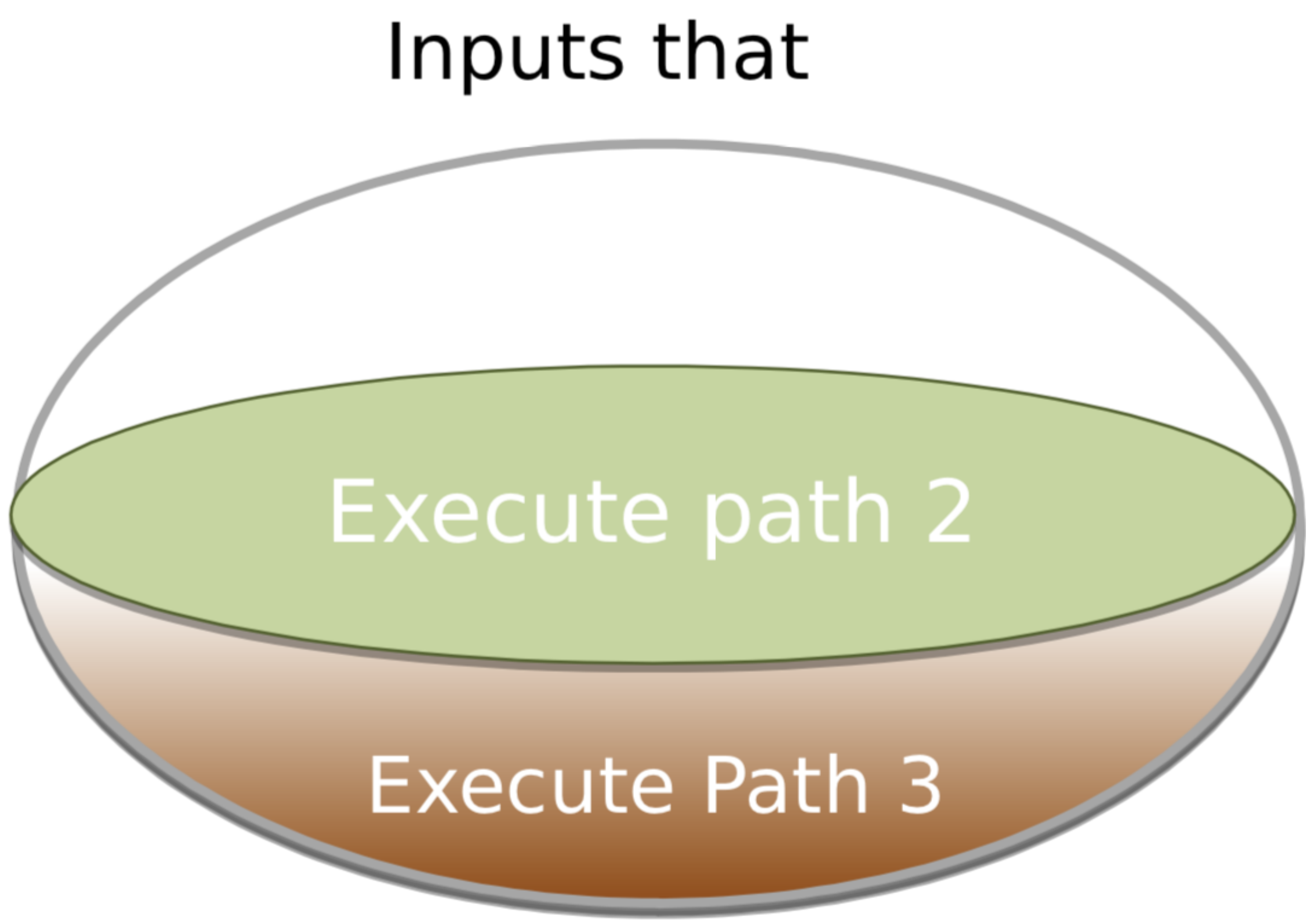
- A technique is efficient if the minimum value is close to expected value
- A technique is **NOT** efficient if minimum << expected value
- There are many cases where minimum << expected for fuzzing

Symbolic Execution

Comparison with fuzzing

- Fuzzing: sample individual inputs from the whole input space
- Symbolic execution: sample inputs from sub input space





Symbolic Execution

Symbolic vs. explicit representation

- What would be the explicit inputs for the follow symbolic formula?
  - x > -4 && x < 4 && x % 2 == 1 && y == x + 3
  - x > -8 && x < 8 && x % 2 == 1 && y == x + 3
  - x % 2 == 1 && y == x + 3

Symbolic Execution

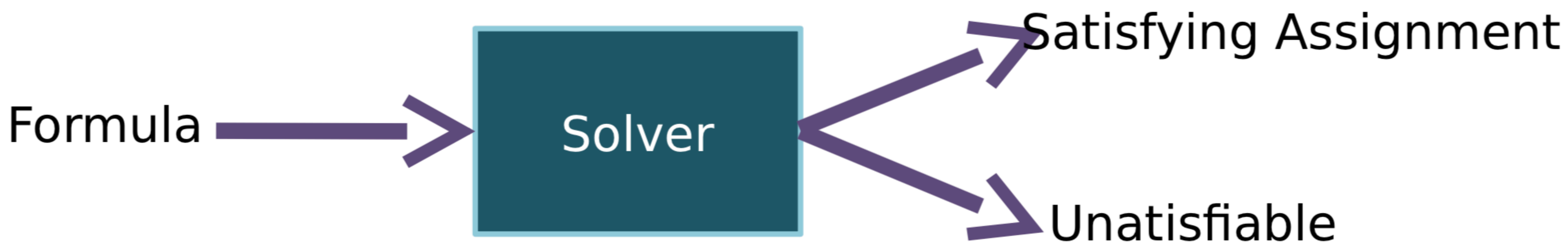
Pros. and Cons. of symbolic representation

- Advantages
  - Can be exponentially smaller than explicit representation of finite sets
  - Can represent infinite sets (e.g., regular expressions)
  - Generic algorithms (e.g., same algorithms for a certain type of formulas)
- Trade-offs
  - Performing basic operations may be expensive
  - Specialized algorithms are required
  - Difficult to predict size of representation

Symbolic Execution

Solvers

- How to sample from a sub-input space of a symbolic representation?
- Solvers: determine if a symbolic formula is satisfiable, if so, provide an example (i.e., satisfying assignments to symbolic variables)
    - An SAT solver is a solver for propositional logic
    - An SMT solver is a solver for formulas in a first-order logic



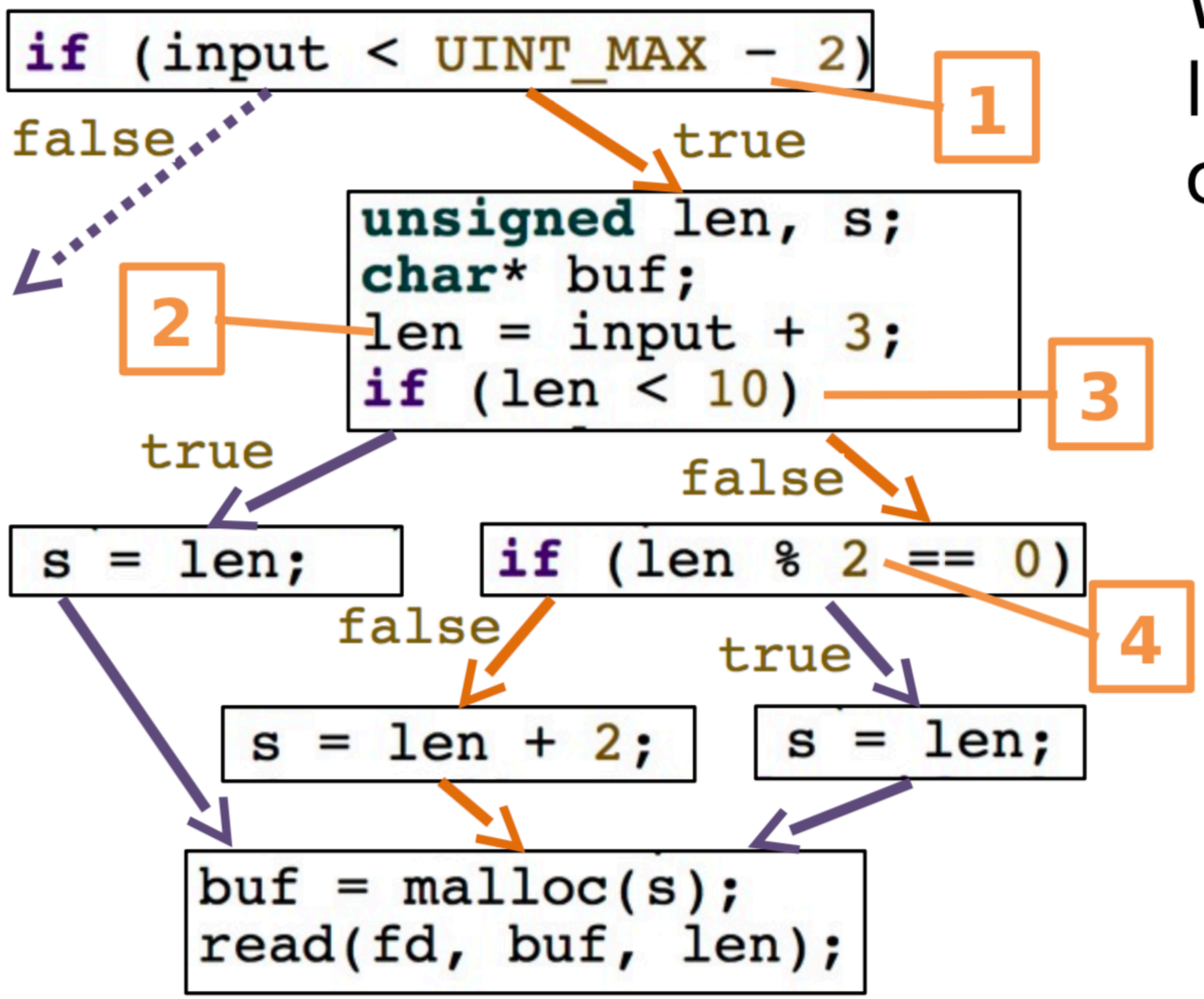
Symbolic Execution

Popular solvers

- Z3: https://github.com/Z3Prover/z3
- CVC4: https://github.com/CVC4/CVC4
- Yices2: http://yices.csl.sri.com/
- STP: https://stp.github.io/

Symbolic Execution

Execution paths as symbolic formulas



Write a formula for the values of len and input that execute the colored path.

1		input < UINT_MAX - 2
2	&&	len == input + 3
3	&&	! (len < 10)
4	&&	! (len % 2 == 0)

Symbolic Execution

Path predicates

- A path predicate encodes the constraints that must be satisfied for a program path to be executed
- To construct a path predicate
  - Rename variables to have unique occurrences (symbolize)
  - Assignments become equalities
  - Branches are themselves, or negated
  - Sequence is conjunction
  - Feasibility of a path == satisfiable of the path predicates

Symbolic Execution

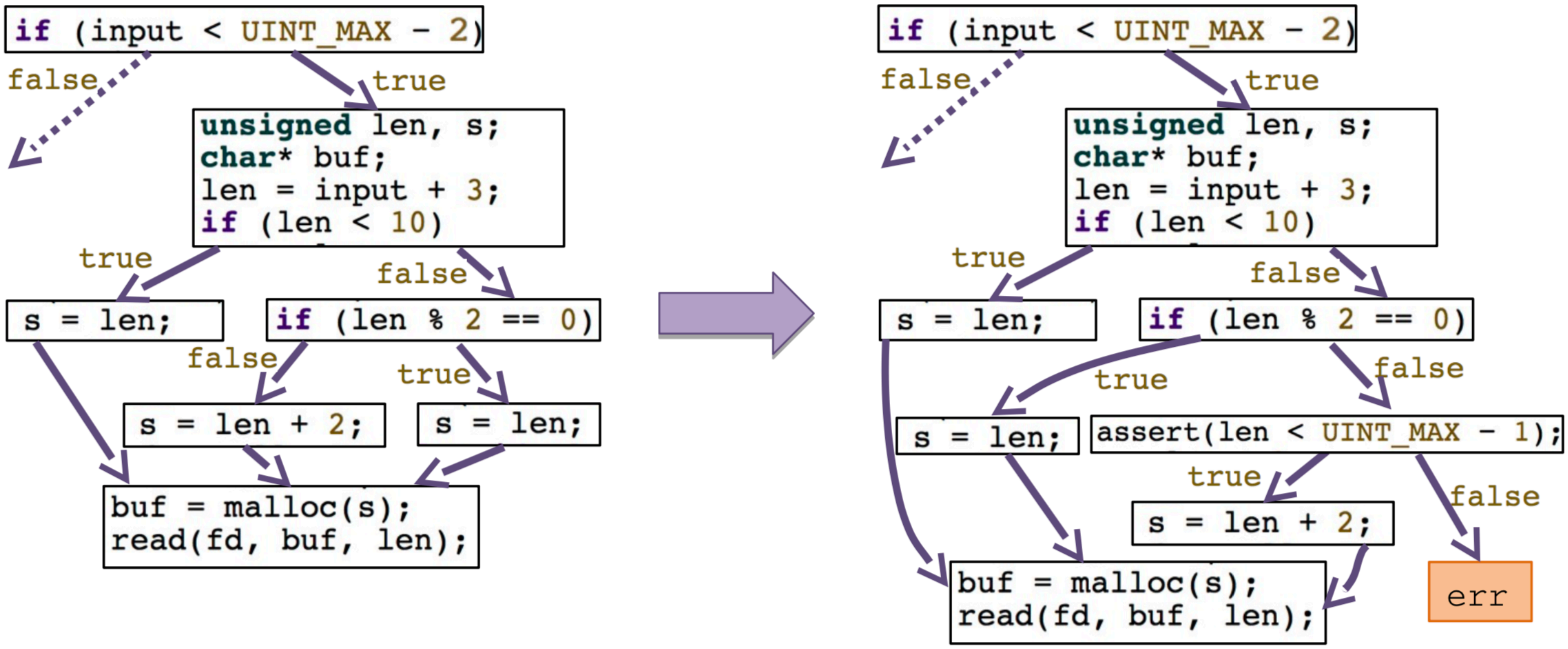
Finding a bug

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else {
      assert(len < UINT_MAX - 1);
      s = len + 2;
    }
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```

Symbolic Execution

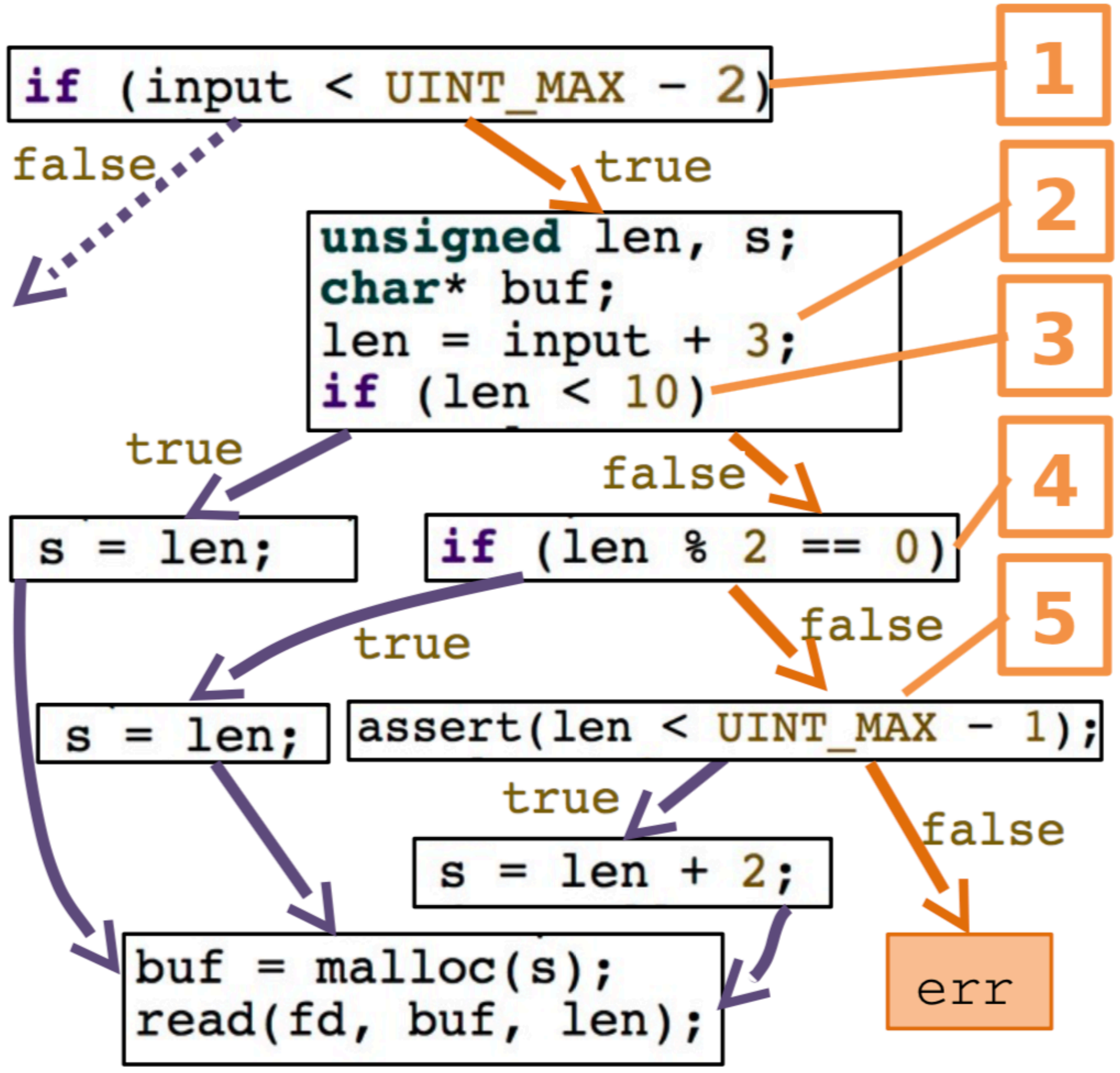
CFG Changes



Symbolic Execution

Path to assertion violation

1		input < UINT_MAX - 2
2	&&	len == input + 3
3	&&	! (len < 10)
4	&&	! (len % 2 == 0)
5	&&	!(len < UINT_MAX - 1)



Symbolic Execution

Bug triggering inputs

- Is the path predicates satisfiable?

```
input < UINT_MAX -2 && len == input + 3 && !(len < 10) && !(len % 2
== 0) && !(len < UINT_MAX – 1)
```

- Yes! When `input == UINT_MAX - 3`

To Learn More

- CS182 and CS206
- Software Testing: From Theory to Practice: https://sttp.site/
- The Fuzzing Book: https://www.fuzzingbook.org/